	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorio de Computacion Salas A y B

Profesor(a): Dávila Pérez René Adrian

Asignatura: Programación orientada a objetos

Grupo: 01

No de practica(s): Proyecto 2

Integrante(s): 322001549, 322006829, 322125533, 425131152, 322103351

No de lista o brigada: 07

Semestre: 2026-1

Fecha de entrega: Viernes 31 de ocutubre 2025

Observaciones:

Calificacion:

Índice

1. Introducción

En el desarrollo de sistemas orientados a objetos, una de las principales necesidades es representar entidades del mundo real de forma estructurada, reutilizable y extensible. En el contexto laboral, los empleados pueden tener diferentes esquemas de remuneración, lo que plantea el reto de diseñar una arquitectura que permita modelar sus características comunes y particulares sin comprometer la claridad ni el escalado del código.

1.1. Planteamiento del problema

Se debe representar distintos tipos de empleados —asalariados y por comisión— dentro de una misma jerarquía de clases, asegurando que cada uno mantenga sus atributos específicos y reglas de negocio (como validaciones de salario o comisiones), mientras comparten una estructura común que permita su tratamiento polimórfico. Es necesario establecer una clase base que encapsule los datos esenciales de todo empleado, como nombre, apellido y número de seguro social, y que defina un comportamiento abstracto para el cálculo de ingresos, el cual será implementado por las subclases según sus particularidades.

1.2. Motivación

Aplicar los principios de la programación orientada a objetos —como la abstracción, herencia y polimorfismo— para construir un sistema robusto y flexible que permita representar empleados con distintos esquemas de pago. Además, se busca fomentar buenas prácticas como el encapsular atributos, la validación de datos y la reutilización de código mediante constructores heredados. Este enfoque no solo mejora la funcionamiento del sistema, sino que también facilita su expansión futura para incluir otros tipos de empleados, como por horas o mixtos.

1.3. Objetivos

El proyecto debe cumplir los siguientes objetivos (requerimientos):

- Se implementa clase abstracta Empleado para representar el concepto general de un empleado. Los atributos de la clase abstracta deben ser privados, o protegidos o finales, que correspondan con los datos de nombre, apellido paterno y numero de seguro social, los cuales van a permitir la extensión de un constructor definido. Respecto a métodos debe tener toString y un método abstracto ingresos los cuales se van a sobre escribir por las subclases.
- La clase heredada EmpleadoAsalariado, cuyo atributo privado extendido es salario semanal. Mediante herencia se va a comunicar con el constructor de la clase abstracta para extender los datos establecidos.
- Respecto a métodos de la clase EmpleadoAsalariado, en el set con el control de que no puede existir salarios con valores negativos y va a sobreescribir ingresos y toString.
- La clase heredada EmpleadoPorComision, cuyos atributos privados extendidos son ventas netas y tarifa de comisión. Mediante herencia se va a comunicar con el constructor de la clase abstracta para extender los datos establecidos.
- Respecto a métodos de la clase EmpleadoPorComision, en el set con dos controles, uno para impedir ventas netas negativas y otro para establecer un rango de tarifa de comisión que no sea menor de 0.0 o mayor a 1.0, en la sobre escritura de ingresos se debe hacer el calculo de la tarifa de comisión por las ventas netas, y sobre escribir el método toString.

- Una clase MainApp donde se probara mediante polimorfismo los tipos de empleado especificados en los puntos anteriores.

2. Marco Teórico

2.1. Clase

Una clase en Java define un nuevo tipo de datos que puede incluir campos (variables) y métodos para definir el comportamiento de los objetos creados a partir de la clase: [1]

```
1
2  class ClassName {
3      // Fields
4      dataType fieldName;
5
6      // Constructor
7      ClassName(parameters) {
8          // Initialize fields
9      }
10
11     // Methods
12     returnType methodName(parameters) {
13         // Method body
14     }
15 }
```

De donde:

- ClassName: El nombre de la clase.
- fieldName: Variables que contienen el estado de la clase.
- Constructor: Método especial utilizado para inicializar objetos.
- methodName: Funciones definidas dentro de la clase para realizar acciones.

2.2. Objeto

Un objeto es una instancia de una clase. Se crea utilizando la palabra clave new seguida del constructor de la clase: [1]

```
1
2
3  ClassName objectName = new ClassName(arguments);
```

De donde:

- objectName: El nombre del objeto.
- arguments: Valores pasados al constructor para inicializar el objeto

A continuación un ejemplo incluyendo ambos conceptos:

```
1
2  class Car {
3      // Fields
4      String color;
5      int year;
6
7      // Constructor
8      Car(String color, int year) {
9          this.color = color;
10         this.year = year;
11     }
12
13     // Method
14     void displayInfo() {
15         System.out.println("Car color: " + color + ", Year: " +
       year);
16     }
17 }
18
19 public class Main {
20     public static void main(String[] args) {
21         // Creating an object
22         Car myCar = new Car("Red", 2020);
23         myCar.displayInfo(); // Output: Car color: Red, Year: 2020
24     }
25 }
```

En este ejemplo, la clase Car tiene los campos color y year, un constructor para inicializar estos campos, y un método displayInfo() para mostrar los detalles del coche. Se crea un objeto myCar utilizando la clase Car.

2.3. Constructor

En Java, un constructor es un método especial utilizado para inicializar objetos. A diferencia de los métodos normales, los constructores se invocan cuando se crea una instancia de una clase. Tienen el mismo nombre que la clase y no tienen tipo de retorno. Los constructores son esenciales para establecer los valores iniciales de los atributos del objeto y prepararlo para su uso.[2]

Tipos de constructores:

1. Constructor por defecto: El compilador Java proporciona automáticamente un constructor por defecto si no se definen constructores explícitamente en la clase. Inicializa el objeto con valores por defecto.
2. Constructor sin argumentos: Un constructor sin argumentos es definido explícitamente por el programador y no recibe ningún parámetro. Es similar al constructor por defecto, pero puede incluir código para inicializar de forma personalizada.[2]
3. Constructor parametrizado: Un constructor parametrizado acepta argumentos para inicializar un objeto con valores específicos. Esto permite inicializar de forma más flexible y controlada de los atributos de los objetos.[2]
4. Copiar Constructor: Un constructor de copia se utiliza para crear un objeto nuevo como copia de un objeto existente. Java no proporciona un constructor de copia por defecto; sin embargo, puede implementarse manualmente.[2]

A continuación sus sintaxis de constructor por defecto:

```
class ClassName {
    // Constructor
    ClassName() {
        // Initialization code
    }

    // Parameterized Constructor
    ClassName(dataType parameter1, dataType parameter2) {
        // Initialization code using parameters
    }

    // Copy Constructor
    ClassName(ClassName obj) {
        // Initialization code to copy attributes
    }
}
```

Ejemplo de constructor:

```
public class Car {
    String model;
    int year;

    // Default Constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    public static void main(String[] args) {
        Car car = new Car();
        System.out.println("Model: " + car.model + ", Year: " +
    car.year);
    }
}
```

En este ejemplo, la clase Car tiene un constructor por defecto que inicializa los atributos model y year con valores por defecto. Cuando se crea un objeto Car, se le asignan estos valores por defecto. [2]

Ejemplo de constructor parametrizado:

```
public class Car {
    String model;
    int year;

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public static void main(String[] args) {
        Car car = new Car("Toyota", 2021);
        System.out.println("Model: " + car.model + ", Year: " +
    car.year);
    }
}
```

Ejemplo de sobrecarga de constructores:

```java
public class Car {
    String model;
    int year;

    // No-Argument Constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Honda", 2022);
        System.out.println("Car1 -> Model: " + car1.model + ",
    Year: " + car1.year);
        System.out.println("Car2 -> Model: " + car2.model + ",
    Year: " + car2.year);
    }
}
```

Este ejemplo demuestra la sobrecarga de constructores en la clase Car, donde se definen tanto un constructor sin argumentos como un constructor parametrizado. Esto permite crear objetos con valores predeterminados o específicos.[2]

2.4. Métodos, parámetros y argumentos

2.4.1. Métodos

Los métodos son bloques de código que se utilizan para realizar tareas específicas. Cada método tiene un nombre que lo identifica y puede recibir una serie de parámetros, los cuales son variables que proporcionan información necesaria para que el método realice su tarea. Estos parámetros actúan como entrada para el método, permitiéndole operar con los valores recibidos.[3]

Dentro del cuerpo del método, se pueden realizar diversas operaciones, como cálculos matemáticos, manipulación de datos o incluso llamadas a otros métodos. Una vez que el método ha completado sus operaciones, puede devolver un resultado utilizando la palabra clave return seguida del valor que se desea devolver.[3]

La capacidad de reutilizar código es una de las ventajas clave de los métodos en Java. Al definir un método una vez, podemos llamarlo en diferentes partes de nuestro programa, evitando la repetición innecesaria de código. Esto no solo simplifica nuestro código, sino que también facilita su mantenimiento y permite una mayor modularidad en nuestra aplicación.[3]

```java
public tipo_de_dato_devuelto nombre_metodo(tipo_de_dato parametro1
    , tipo_de_dato parametro2) {
    // codigo a ejecutar
    return resultado;
}
```

2.4.2. Parámetros

Son las variables que se definen en la declaración de un método y sirven para recibir información externa. Actúan como entradas que el método necesita para ejecutar sus instrucciones.[3]

```
public void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "!    Cmo    est s?");
}
```

En el método denominado “saludar” se recibe un parámetro de tipo String llamado “nombre”. Dentro del cuerpo del método, se imprime una cadena de saludo que incluye el valor del parámetro “nombre”. Al llamar a este método, debemos proporcionar un argumento de tipo String que se asignará al parámetro “nombre”.

2.4.3. Argumentos

Son los valores concretos que se pasan a un método cuando este es llamado. Estos valores corresponden a los parámetros definidos en el método. Una vez que hemos declarado un método, podemos llamarlo desde otro lugar de nuestro código. La llamada de un método se realiza utilizando su nombre seguido de paréntesis, y se pueden pasar los argumentos correspondientes si es necesario.[3]

```
int resultado = sumar(5, 3); // Llamada a un metodo con retorno
```

```
public int sumar(int a, int b) {
    return a + b;
}
```

2.5. Encapsulamiento

La encapsulación es un principio fundamental de la programación orientada a objetos (POO) en Java que consiste en agrupar los datos (variables) y los métodos (funciones) que operan sobre los datos en una sola unidad, conocida como clase. Restringe el acceso directo a algunos de los componentes de un objeto y puede evitar la modificación accidental de datos. [4]

La finalidad de la encapsulación es la siguiente:

- Ocultar datos: La encapsulación permite ocultar al exterior la representación interna de un objeto. Sólo se exponen los detalles necesarios a través de una interfaz pública.
- Mayor flexibilidad: Al controlar el acceso a los campos de una clase, puedes cambiar la implementación interna sin afectar al código externo que utiliza la clase.
- Mantenimiento mejorado: La encapsulación ayuda a mantener el código manteniendo los campos privados y proporcionando métodos getter y setter públicos para modificar y ver los campos.

Para lograr la encapsulación en Java:

1. Declara las variables de clase como private.
2. Proporciona los métodos getter y setter de public para acceder y actualizar el valor de una variable privada.

A continuación un ejemplo:

```java
public class Student {
    private String name;
    private int age;

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

En este ejemplo, la clase Student tiene los campos privados name y age. Se proporcionan métodos públicos getter y setter para acceder a estos campos y modificarlos. El configurador de age incluye una comprobación de validación básica.

A continuación un segundo ejemplo con validación:

```java
public class BankAccount {
    private double balance;

    // Getter method for balance
    public double getBalance() {
        return balance;
    }

    // Setter method for balance with validation
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

La clase BankAccount encapsula el campo balance. Proporciona los métodos deposit y withdraw para modificar la balanza, garantizando que sólo se realizan operaciones válidas.

2.6. Herencia

La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos, por medio del cual una clase se construye a partir de otra. Una de sus funciones más importantes es proveer el polimorfismo. Relaciona las clases de manera jerárquica; una clase padre, superclase o clase base sobre otras clases hijas, subclases o clase derivada. [5].

Los descendientes de una clase, heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables, además de crear los suyos propios. En Java, sólo se permite la herencia simple, o dicho de otra manera, la jerarquía de clases tiene estructura de árbol. El punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases. Para especificar la superclase se realiza con la palabra *extends*; si no se especifica, se hereda de Object. [5].

2.6.1. Definición de superclase

Comienza definiendo una clase base, también llamada superclase, que contiene los atributos y métodos comunes que se desean compartir con las clases derivadas [6].

```
1
2  public class Animal {
3      String nombre;
4      public void comer() {
5         System.out.println("El animal esta comiendo.");
6      }
7  }
```

2.6.2. Creación de clase derivada

Posteriormente, crea una nueva clase que herede de la clase base usando la palabra clave *extends*.

```
1
2  public class Perro extends Animal {
3      public void ladrar() {
4          System.out.println("El perro esta ladrando.");
5      }
6  }
```

Ahora es posible crear objetos de la clase derivada, permitiendo el acceso a los atributos y métodos en clase base o derivada.

```
1
2  public class Main {
3      public static void main(String[] args) {
4          Perro miPerro = new Perro();
5          miPerro.nombre = "Bobby";
6          miPerro.comer();  // Metodo de la clase base
7          miPerro.ladrar(); // Metodo de la clase derivada
8      }
9  }
```

En este ejemplo, la clase Perro hereda de la clase Animal, por lo que un objeto de la clase Perro también puede acceder a los métodos y atributos de la clase Animal. La herencia permite compartir y reutilizar funcionalidades comunes [6].

2.6.3. Clase abstracta

Una clase abstracta es una clase que no es posible instanciar directamente. Es decir, no se crea un objeto a partir de ella, pero sí se usa para definir un "esqueleto" de lo que otras clases deben hacer. Este esqueleto puede tener métodos abstractos (que las subclases deben implementar) y métodos con comportamiento ya definido.[7]

Imaginemos que tenemos una clase abstracta Persona, y que todas las personas pueden andar y correr, pero cómo lo hacen depende de quién sea. El método para andar y el de correr no van a ser iguales para un deportista que para un informático, ¿verdad? Así que vamos a dejar esos métodos como abstractos, y haremos que las subclases los implementen a su manera.[7]

```
1
2  abstract class Persona {
3      // M todo abstracto para andar (las subclases deben
       implementarlo)
4      abstract void andar();
5
6      // M todo abstracto para correr
7      abstract void correr();
8  }
```

Aquí hemos se define la clase abstracta Persona. Los métodos andar y correr son abstractos, lo que significa que las subclases, como Deportista e Informático, tendrán definirán cómo deben comportarse. Un Deportista, como es lógico, tiene su propio estilo tanto para andar como para correr, y vamos a implementarlo.

```
1
2  class Deportista extends Persona {
3      @Override
4      void andar() {
5          System.out.println("El deportista anda r pido.");
6      }
7
8      @Override
9      void correr() {
10         System.out.println("El deportista corre a gran velocidad."
       );
11     }
12 }
```

Aquí, el deportista camina rápido y, por supuesto, corre aún más rápido.

Por otro lado, un Informático probablemente tenga un estilo de vida más sedentario, y esto también se refleja en cómo camina y corre.[**?**]

```
1
2  class Informatico extends Persona {
3      @Override
4      void andar() {
5          System.out.println("El inform tico anda lentamente.");
6      }
7
8      @Override
9      void correr() {
10         System.out.println("El inform tico prefiere no correr.");
11     }
12 }
```

Con las subclases implementadas, se analiza su comportamiento cuando son utilizadas.

```
public class Main {
    public static void main(String[] args) {
        Persona deportista = new Deportista();
        deportista.andar();    // Output: El deportista anda
    r pido.
        deportista.correr();   // Output: El deportista corre a
    gran velocidad.

        Persona informatico = new Informatico();
        informatico.andar();   // Output: El inform tico anda
    lentamente.
        informatico.correr();  // Output: El inform tico prefiere
     no correr.
    }
}
```

```
El deportista anda rapido.
El deportista corre a gran velocidad.
El informatico anda lentamente.
El informatico prefiere no correr.
```

Las clases abstractas en Java son una excelente herramienta cuando es necesario definir comportamientos comunes, pero que puedan ser personalizados por cada subclase. En este ejemplo, anteriormente se observó cómo una clase abstracta *Persona* obliga a que sus subclases, Deportista e Informático, implementen a su manera los métodos andar y correr.[7]

Con esta estructura, se garantiza que todas las subclases cumplan con ciertos requisitos, pero sin perder flexibilidad a la hora de definir cómo deben hacerlo.[7]

2.7. Polimorfismo

El polimorfismo es la capacidad de que un mismo mensaje funcione con diferentes objetos. Es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo concreto de objetos sobre el que se trabaja, el método opera sobre un conjunto de posibles objetos compatibles.[5].

Para crear una clase base, se deben definir los métodos que desean que las clases derivadas implementen de forma específica, por ejemplo:

```

public interface Animal {
   void hacerSonido();
}
```

Posteriormente se crean clases concretas que implementen la interfaz o hereden de su clase base. Cada base debe proporcionar su propia implementación del método definido anteriormente.

```
1
2  public class Perro implements Animal {
3      public void hacerSonido() {
4          System.out.println("El perro ladra.");
5      }
6  }
7
8  public class Gato implements Animal {
9      public void hacerSonido() {
10         System.out.println("El gato maulla.");
11     }
12 }
```

Ahora es posible crear objetos de clases derivadas y tratarlos como si se tratase de objetos de la clase base, por lo que es posible llamar un método común, en el Main de esta manera:

```
1
2  public class Main {
3      public static void main(String[] args) {
4          Animal miMascota = new Perro();
5          miMascota.hacerSonido(); // Llama al metodo del perro
6
7          miMascota = new Gato();
8          miMascota.hacerSonido(); // Llama al metodo del gato
9      }
10 }
```

En este ejemplo, se crean los objetos objetos Perro y Gato, asignados a una referencia de tipo Animal y se llama al método *hacerSonido()* sin necesidad de la implementación real de cada clase. Esto es el polimorfismo: objetos de diferentes clases responden al mismo mensaje (hacerSonido()) de manera específica para cada uno [6]

2.8. Paquetes

Un paquete (package) en Java es un espacio de nombres (namespace) que organiza un conjunto de clases, interfaces, enumeraciones y anotaciones relacionadas. Actúa como una carpeta lógica que agrupa elementos con funcionalidad similar.

Para crear un paquete, debemos utilizar la declaración del paquete agregándola como la primera línea de código en un archivo . [8]

Se recomienda encarecidamente colocar cada nuevo tipo en un paquete. Si definimos tipos y no los colocamos en un paquete, se incluirán en el paquete predeterminado o sin nombre. Usar paquetes predeterminados tiene algunas desventajas:

- Perdemos los beneficios de tener una estructura de paquete y no podemos tener subpaquetes.
- No podemos importar los tipos del paquete predeterminado desde otros paquetes.
- Los ámbitos de acceso protegido y privado del paquete no tendrían sentido.

Por ejemplo, para crear un paquete a partir de www.baeldung.com , invirtámoslo:

```
1  com.baeldung
```

Comencemos definiendo una clase TodoItem en un subpaquete llamado dominio :

```java
package com.baeldung.packages.domain;

public class TodoItem {
    private Long id;
    private String description;

    // standard getters and setters
}
```

Al compilar nuestras clases empaquetadas, debemos recordar la estructura de directorios. Empezando por la carpeta de origen, debemos indicarle a javac dónde encontrar nuestros archivos. [9]

Primero necesitamos compilar nuestra clase TodoItem porque nuestra clase TodoList depende de ella.Comencemos abriendo una línea de comando o terminal y navegando a nuestro directorio de origen. Ahora, compilemos nuestra clase com.baeldung.packages.domain.TodoItem :

```
> javac com/baeldung/packages/domain/TodoItem.java
```

Ahora que nuestra clase TodoItem está compilada, podemos compilar nuestras clases TodoList y TodoApp :

```
>javac -classpath . com/baeldung/packages/*.java
```

Ejecutemos nuestra aplicación usando el nombre completo de nuestra clase TodoApp :

```
>java com.baeldung.packages.TodoApp
```

3. Desarrollo

La implementación realizada pertenece al paquete mx.unam.fi.poo.proyecto2, y se basa en el uso de los principios de herencia y polimorfismo dentro de la programación orientada a objetos.

Se ha definido una clase abstracta llamada **Empleado**, que representa el concepto general de un empleado dentro de una empresa. Esta clase contiene los atributos comunes a todos los empleados, tales como **nombre**, **apellido** y **número de seguridad social (NSS)**. Además, declara un método abstracto denominado **ingresos()**, el cual obliga a las clases derivadas a definir su propia forma de calcular las ganancias o ingresos correspondientes.

A partir de esta clase base, se derivan dos subclases concretas:

- **EmpleadoAsalariado** La clase representa a los empleados que reciben un salario fijo semanal. Implementa el método *ingresos()* devolviendo directamente el valor del salario semanal. También se incluye validación para evitar valores negativos y un método de representación textual que muestra los datos personales junto con el salario.
- **EmpleadoPorComision** Representa a los empleados cuyos ingresos dependen de sus ventas. Implementa el método *ingresos()* multiplicando las ventas netas por la tarifa de comisión. Se agregan controles para validar que la tarifa esté en el rango de 0 a 1 y que las ventas sean valores positivos.

Finalmente, la clase **MainApp** funciona como punto de entrada del programa. En ella se crean instancias de ambos tipos de empleados y se almacenan en un arreglo del tipo Empleado. Gracias al polimorfismo, es posible procesar a todos los empleados de manera uniforme, sin importar si son asalariados o por comisión, utilizando el mismo método **ingresos()** definido en la clase base.

Pruebas realizadas

Las pruebas se realizaron creando instancias de las clases mencionadas y verificando que el cálculo de ingresos se efectúe correctamente según el tipo de empleado.

- **Prueba 1: Entrada:** Empleado asalariado con nombre *Ana López*, NSS *123-456-789*, salario semanal de *$8000.00*. **Salida esperada:** Muestra los datos del empleado junto con los ingresos semanales de *$8000.00*.
- **Prueba 2: Entrada:** Empleado por comisión con nombre *Carlos Ruiz*, NSS *987-654-321*, ventas netas de *$50000.00* y tarifa de comisión de *0.10*. **Salida esperada:** Muestra la información del empleado junto con los ingresos semanales de *$5000.00* (10 % de las ventas).
- **Prueba 3: Entrada:** Empleado asalariado con nombre *Laura García*, NSS *111-222-333*, salario semanal de *$12000.00*. **Salida esperada:** Se muestran los datos del empleado y los ingresos semanales de *$12000.00*.

4. Resultados

Para verificar que el código funciona correctamente después de integrar la herencia, el polimorfismo y el empaquetamiento de clases, el proyecto se compiló y ejecutó desde la terminal para probar las clases **EmpleadoAsalariado** y **EmpleadoPorComisión**. Los comandos utilizados fueron los siguientes:

```
PS C:\Users\alex\OneDrive\Desktop\POO\Proyecto2> javac -d out mx/unam/fi/poo/proyecto2/*.java
PS C:\Users\alex\OneDrive\Desktop\POO\Proyecto2> java -cp out mx.unam.fi.poo.proyecto2.MainApp
```

El primer comando compila todos los archivos `.java` y genera los correspondientes `.class` en la carpeta de salida `out`. Posteriormente, se invoca el `MainApp.class` especificando el paquete raíz mediante la opción `-cp`, lo que permite ejecutar el programa sin errores.

La salida obtenida es la siguiente:

```
EMPLEADOS DE LA EMPRESA


Procesando empleados de forma polimórfica:

Empleado Asalariado:
Empleado: Ana López
NSS: 123-456-789
Salario Semanal: $8,000.00
Ingresos Semanales: $8,000.00

Empleado por Comisión:
Empleado: Carlos Ruiz
NSS: 987-654-321
Ventas Netas: $50,000.00
Tarifa Comisión: 0.10
Ingresos Semanales: $5,000.00

Empleado Asalariado:
Empleado: Laura García
NSS: 111-222-333
Salario Semanal: $12,000.00
Ingresos Semanales: $12,000.00
```

En la ejecución se observan los siguientes puntos que confirman el comportamiento esperado:

- **Procesamiento polimórfico de empleados.** Para cada empleado se puede observar el método sobrescrito `ingresos()`, demostrando el polimorfismo que fue implementado para cada subclase.
- **Empleado Asalariado.** Para cada empleado asalariado se muestra el nombre, NSS y el salario semanal. El método `ingresos()` devuelve el mismo salario semanal, lo cual coincide con los valores mostrados en pantalla.
- **Empleado por Comisión.** Para los empleados con pago por comisión se reportan las ventas netas y la tarifa de comisión; `ingresos()` calcula $ventas \times tarifa$, lo que se refleja en los montos mostrados (por ejemplo, ventas de `$50,000.00` con tarifa `0.10` producen `$5,000.00`).
- **Formato y encapsulamiento.** Los datos impresos (nombre, NSS, ventas, salario, comisión) se obtienen mediante métodos de acceso, respetando el encapsulamiento.

Con lo anterior, se demuestra que las subclases **EmpleadoAsalariado** y **EmpleadoPorComisión** heredan de la clase base y sobrescriben correctamente su comportamiento; asimismo, la ejecución desde la carpeta `out` garantiza que el empaquetamiento y la estructura de paquetes se encuentran configurados de forma adecuada.

5. Conclusiones

La implementación del sistema de clases para empleados permitió aplicar de forma efectiva los principios de la programación orientada a objetos. A través de la clase abstracta Empleado, se definió una estructura común que facilita la extensión mediante herencia, y se estableció un método abstracto para calcular ingresos, lo que obliga a cada subclase a definir su propia lógica.

Las clases EmpleadoAsalariado y EmpleadoPorComision sobrescriben correctamente los métodos requeridos y aplican validaciones que aseguran datos coherentes, como evitar salarios negativos o comisiones fuera de rango. Esto mejora la confiabilidad del sistema y refleja un buen manejo de reglas de negocio.

En la clase MainApp, se demuestra el uso exitoso del polimorfismo, permitiendo tratar distintos tipos de empleados de forma uniforme. Los resultados en consola confirman que cada objeto responde correctamente a sus atributos y comportamientos, mostrando advertencias cuando hay datos inválidos y calculando los ingresos según el tipo de empleado.

Por lo tanto, el sistema cumple con los objetivos planteados: modelar empleados de forma clara, segura y extensible, utilizando herencia, encapsulamiento y polimorfismo de manera adecuada.

Referencias

[1] DataCamp. (2025) Clases y objetos en java. Accedido: 16 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/classes-and-objects

[2] ——. (2025) Java constructors. Accedido: 16 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/constructors

[3] A. Barragán. (2023) Introducción a java: Métodos, parámetros y argumentos. Publicado: 24 de octubre de 2023; Accedido: 16 de septiembre de 2025. [Online]. Available: https://openwebinars.net/blog/introduccion-a-java-metodos-parametros-y-argumentos/

[4] DataCamp. (2023) Encapsulation in java. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/encapsulation

[5] J. B. Bermúdez, *Programación Orientada a Objetos en Java*, Documento académico en línea, Escuela Técnica Superior de Ingenieros en Sistemas Informáticos, Madrid, España, 2013, curso 2013-2014, Máster en Ingeniería Web (MIW). [Online]. Available: https://www.etsisi.upm.es/sites/default/files/curso$_2$013$_1$4/$MASTER/MIW.JEE.POOJ.pdf$

[6] J. L. Blasco, "Introducción a poo en java: Herencia y polimorfismo," 2023, publicado: 17 de noviembre de 2023, Accedido: 2025-10-08. [Online]. Available: https://openwebinars.net/blog/introduccion-a-poo-en-java-herencia-y-polimorfismo/

[7] C. Álvarez Caules. (2024) Java: Clases abstractas y el polimorfismo. Artículo en línea. Arquitectura Java. España. [Online]. Available: https://www.arquitecturajava.com/java-clases-abstractas-y-el-polimorfismo/

[8] Oracle. (2025) What is a package? Accedido: 30 de septiembre de 2025. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/concepts/package.html

[9] Baeldung. (2024) Guide to java packages. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.baeldung.com/java-packages